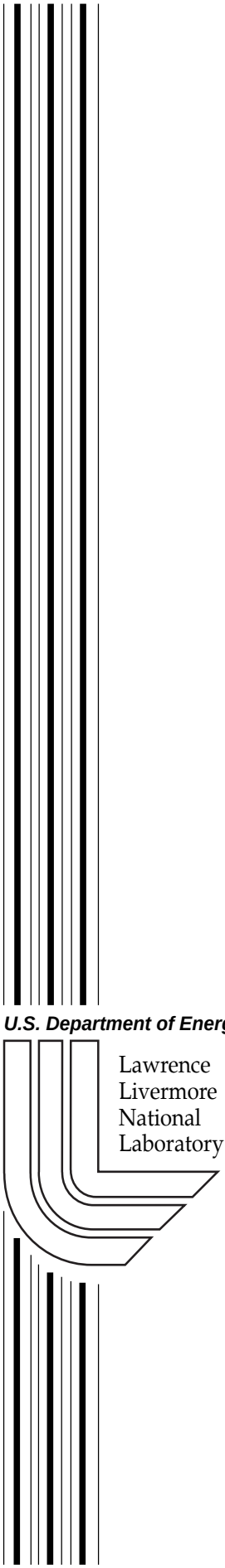
Preprint
UCRL-JC-151792

A Source-to-Source Architecture for User-Defined Optimizations

M. Schordan, D. Quinlan

This article was submitted to *Joint Modular Languages Conference 2003, Klagenfurt, Austria*
08/25/2003 – 08/27/2003



February 6, 2003

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

A Source-To-Source Architecture for User-Defined Optimizations

Markus Schordan* Dan Quinlan

February 6, 2003

Abstract

The performance of object-oriented applications often suffers from the inefficient use of high-level abstractions provided by underlying libraries. Since these library abstractions are user-defined and not part of the programming language itself only limited information on their high-level semantics can be leveraged through program analysis by the compiler and thus most often no appropriate high-level optimizations are performed.

In this paper we outline an approach based on source-to-source transformation to allow users to define optimizations which are not performed by the compiler they use. These techniques are intended to be as easy and intuitive as possible for potential users; i.e., for designers of object-oriented libraries, people most often only with basic compiler expertise.

Introduction

In the engineering of language based software, there are two extreme approaches. One is to just use a regular programming language, the other is to use a very high-level formalism, such as attribute grammars. The technique we present was designed to fill a place in between these extremes, and to attempt to bridge the gap.

The user can be expected to have reasonable knowledge of the language in which the program is written which he desires to optimize. Allowing him to express the optimization in this language eliminates the barrier that needs to be overcome when learning a new language with different syntax and semantics. But at the same time, what he is required to express should only focus on the aspects of the program transformation necessary to perform the optimization but not significantly more. Any non-trivial program transformation requires knowledge about the structure of a program. Therefore we use as computation model structural induction which is a special case of attributed grammars and requires only one pass over the program fragment that is to be transformed. New information about the program is computed by an attribute evaluation mechanism. New program fragments, those that are added or replace existing program fragments, can be expressed as source-strings. This is accomplished by reinvoking the front-end for program fragments to be added and for which an intermediate representation is created *during* a structural induction on the input program. Once the program has been transformed into a more efficient program it can be unparsed (as source code) and compiled with the vendor compiler.

In our case, the language for which our framework, ROSE [1], allows to express optimizations is C++. Hence, we allow to express the optimizations in the same

*Centre for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551, USA (schordan1@llnl.gov).

language, in C++. The user needs to be aware of a structural representation of source code constructs, i.e. an abstract syntax tree (AST), and in which order this representation is traversed. By traversing the AST any source-strings can be computed by evaluating attributes. To make the modification of the AST convenient for the user, our framework automatically extends such a source-string with all necessary declarations and runs it as completed program through the front-end and extracts the AST fragment. This AST fragement is then inserted at the specified AST location and all internal data structures are updated. Additionally the user can unparse the AST at any given AST node and obtain a string representing the subtree with this node as root node.

The AST is annotated with all static type information, scoping information, and symbol table information. This information is also available for any program fragment that is inserted into the AST. No action is required within a transformation to update dependencies in the AST when replacing, adding, or deleting program fragments. All semantic information is always updated by the underlying system. The order in which the AST is traversed is defined by its structure but semantic information about the program can be used when deciding if and where a transformation is to be applied. This is particuarly useful when transforming programs with user-defined types.

To have semantic information available is essential for high-level optimizations. We have shown in [1, 2] that the performance penalty of high-level abstractions can be overcome by source-to-source transformations and presented promising results for user-defined abstractions as they are used in practice. We are able to get a similar performance for C++ code with high-level abstractions as for low-level C code, and by using the restrict keyword we can also match the performance of Fortran programs. This results encouraged us to build a source-to-source architecture that allows to automate many optimizations that otherwise would have to be done by hand to achieve the desired level of performance with today's commercial compilers.

In the remaing sections we describe the architecture we developed in detail and how the definition of program transformations is simplified by the capabilities of the discussed architecture. We also give an overview of the infrastructure available for debugging transformations and visualization of the intermediate representation. Eventually we discuss performance aspects of the implementation.

Source-To-Source Architecture

In our source-to-source architecture the input language and output language is the same. Therefore an output of the back-end can always be an input to the front-end. Similar for code fragments, the architecture allows to obtain the corresponding intermediate representation fragment (IR-fragment) for a given source-fragment by reinvoking the front-end. Any IR-fragment can be unparsed to a source-fragment by (re)invoking the back-end. We call this "bending" the front-end and back-end. Both operations can be performed when processing the indermediate representation and evaluating attributes. Hence, IR-fragments as well as source-fragments, can be computed as attributes and used to describe which code is to be inserted and/or replaced in the IR. This interplay allows to use either representation when defining a transformation.

We did not modify the front-end to allow to parse source-fragments. We use a commercial front-end, the EDG C++ front-end [3]. Instead, we extend a source-fragment automatically to a complete program which can be parsed by the front-end and extract the corresponding IR-fragment from the generated IR. These two translation processes are accomplished by the Fragment Concatenator and Fragment Extractor.

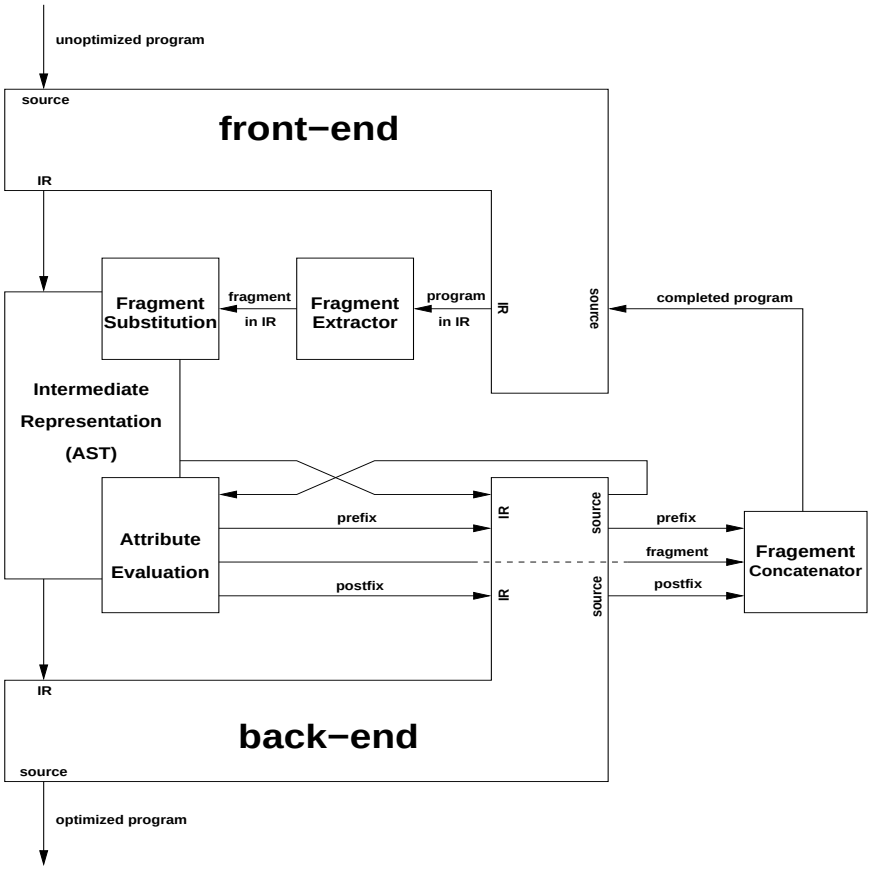


Figure 1: Source-To-Source architecture with bended front-end and back-end

The relationship between the Fragment Concatenator and Fragment Extractor is that the Fragment Extractor strips off all IR code that represents source code that needed to be added by the Fragment Concatenator to complete a given source-fragment to a program that can be parsed by the front-end. The source-prefix and source-postfix that are needed to extend a source-fragment are computed by the system. The user only needs to provide the source-fragment.

Synthesized attributes and (restricted) inherited attributes can be used to define a transformation. The traversal order is fixed, which we therefore call an evaluation scheme for attributes. An attribute can be of arbitrary type because we provide generic interfaces for the attribute evaluation.

Intermediate Representation

The intermediate representation we use is an annotated abstract syntax tree (AST), which is an imploded Parse Tree annotated with all static type information for each expression, symbol tables for every scope, and cross-references to definitions from declarations where useful.

Usually, an object-oriented abstract syntax tree represents lists of language elements as a collection of objects, and single language elements as single objects. This holds for our design for all nodes.

Another important aspect which also influenced the design of the AST is the ease of access of computed attributes. Since the evaluation order is fixed the computation of attributes is fixed as well and the access of an attribute access can be done by

name. To make the access as simplicistic as possible we developed an AST design that allows to use a straight-forward naming scheme based only on the node type (name) and the names of pointers to the successor nodes of an AST node. Note that an inherited and synthesized attribute can be of arbitrary type.

Therefore we have redesigned the SAGE II representation [4] to conform to the following properties, now called SAGE III:

We have two different kinds of nodes with respect to the successor information of the AST:

- Container nodes: A node only has a single container of pointers to successor node objects.
- Non-Container nodes: A node only has several single pointers to successor node objects.

We changed the interfaces of 47 node types and added 5 new node types of the SAGE II representation. With these constraints on all nodes also the visualization of the AST was simplified, thus making it easier to communicate the structure and transformations on it to the user.

Attribute Evaluation

Perhaps the most straightforward style of computing a value over a tree is by structural induction. The result of a tree is computed as a function of the results of its subtrees.

Structural induction is a special case of attribute grammars, where there is only one pass, computing only synthesized attributes. Each node is decorated with a number of attributes, of which the value is computed from the values of the subterms of the node (synthesized attributes) or from the encompassing node (inherited attributes). An evaluation scheme walks the tree to compute all the attributes.

We provide four different generic interfaces: a "classic" traversal of the AST without attribute evaluation, and a top-down, bottom-up, and a combined top-down-bottom-up evaluation scheme with inherited attributes for top-down and synthesized attributes for bottom-up evaluation. Additionally we allow to attach user-defined attributes to any AST node (AST attributes). These AST attributes can be used to share information between different transformantions and allow to build object-oriented hierarchies of transformations and build abstractions and nested evaluations for any part of a transformation.

Fragment Concatenator and Extractor

The Fragment Concatenator concatenates prefix, fragment, and postfix and wrapps code around it to generate a program that can be parsed by the front-end. The Fragment Extractor unwrappes code, and strips off prefix and postix, to obtain an AST fragment which represents the source fragment.

A source-string is called a source-fragment if all of the following holds

- The fragment can be represented by an AST subtree with a single root node.
- All arguments of (unary, binary, and ternary) operators are present.
- New declarations must be complete.

A fragment can always be completed by an automatically generated prefix, postfix, and wrapper code to form a program that can be parsed by the front-end.

```
MySynthesizedAttribute
evaluateSynthesizedAttributeOfSgWhileStatement(SgNode* node,
                                               MyInheritedAttribute ia,
                                               SubTreeSynthesizedAttributes c) {
  return c[SgWhileStmt_condition] + c[SgWhileStmt_body];
}

MySynthesizedAttribute
evaluateSynthesizedAttributeOfSgBasicBlock(SgNode* node,
                                           MyInheritedAttribute ia,
                                           SubTreeSynthesizedAttributes c) {
  MySynthesizedAttribute s;
  if (ia.useThisBlock) {
    for(SubTreeSynthesizedAttributes::iterator i=c.begin(); i!=c.end(); i++) {
      s += *i;
    }
  }
  node->setAttribute("mynewattribute",new MyAstAttribute(s)); // add AST attribute
}
```

Figure 2: Example of C++ user-code to define the evaluation of synthesized attributes for SgWhileStatement and SgBasicBlock

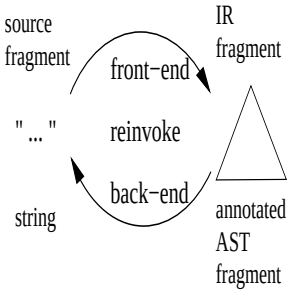


Figure 3: A source-fragment can always be translated into an AST fragment by reinvoking the front-end and any AST fragment can always be translated into a source-fragment by reinvoking the back-end

The source-prefix is defined by the sequence of declarations and opening scopes from the first language construct in the program to the start of the source-fragment. The postfix is defined by the closing scopes that are required to complete it to obtain a correct program. Note that not every source-string is a source-fragment but that the set of all source-fragments is a strict subset of all source-strings.

Because a source-fragment can always be completed and parsed by the front-end we can always obtain the corresponding AST fragment for a given source-fragment.

For unparsing an AST fragment only the first of the three conditions must hold, i.e. unparsing can be started for any node of the AST.

Note that a source-prefix is not represented by subtree in general but a source-prefix, which is a fragment, can always be represented by an AST fragment which is generated when reinvoking the front-end.

Since both representations, source-fragments and AST fragments can always be translated to the one or other, both can be used interchangably allowing the user to use source-strings where it is convenient to define new code fragments by using source code patterns or traversing an AST subtree to compute attributes. Attributes can be of any type, including source-strings.

Fragment Substitution

An AST fragment which is obtained from the Fragement Extractor corresponds to the source-fragment which is the input to the Fragement Concatenator. The location in the AST where this new AST fragment needs to be inserted is defined by the root node of the AST subtree to be replaced. This root node can be defined relativ to a current location in a traversal on the AST or absolute as a direct pointer to the node.

A relative position of the substitution root node can only be specified for locations which are on the path of the bottom-up-traversal from the current node to the AST root node. But the pointer to such a node is not required, instead different cases such as outer scope, file scope can be defined and are provided by our infrastructure. This simplifies moving code betweeen scopes/blocks as necessary for loop hoisting, loop fusion, etc.

The use of relative positions when generating lower-level code has been most useful in the source-to-source translators we have implemented so far. This is an example where attributed grammers are more difficult to use because an attribute would be required to propagate this information and the location where this code needs to be used would have to be associated with a rule. For example, when we want to move code, we can specify the target by means of outer loop scope, (current) function scope, current or other file scope, etc., from the current position. This functionality is implemented internally by using the attribute mechanism but hidden from the user, and by ensuring an order of modifications on the AST as discussed in the next section.

Program Transformations

The intermediate representation is rewritten by a transformation. The underlying system ensures that if only strings are used to specify replacements, all internal data structures are updated. The user does not need to modify any pointers of the AST.

A transformation is the behaviour of an object where at most two virtual functions need to be implemented, one for evaluating inherited attributes, and one for evaluating synthesized attributes.

That the AST is rewritten is not (visible as) a side-effect of the evaluate functions which the user implements. A transformation is started when the traverse function on a transformation object is invoked but the AST does not get modified before the traversal on the respective subtree is finished. Clearly, any modification of AST is performed after the traversal on a respective subtree has completed. The function calls to move/copy/insert/replace fragments of the AST are buffered and no modification takes place before all user-defined functions (at most two) on a node have returned control to the system. This is accomblished by storing the relative position where the user specified to insert a source-fragment and by buffering the source-fragments, until the node where the insert operation actually needs to take place is reached.

In general, any source-fragment based rewrite of the AST ensures that the semantics of a single transformation are that no side effect on the AST becomes visible before the transformation is finished, i.e. like a side-effect free function with respect to the AST. However, side-effects like writing to stdout, can take place because it is ordinary C++ code but as long as only attributes and source-fragments are used to define transformations the user is on the safe side.

For a sequence of transformations this is not the case. This allows to combine different specialized transformations, and to apply transformations iteratively. To

some extent rewrite systems, by using pattern matching (AST read-only traversals, or AST "queries"), can be implemented but our system is less efficient than comparable rewrite systems for similar tasks.

Debugging and visualization

We allow to dump out any subtree of the intermediate representation in two formats. A visualization of the AST is generated by using the tool dot. We dump out a textual description of the AST and dot translates this description into a visualization, allowing to specify a whole bunch of different formats like postscript, gif, or jpeg. Although this helps the user to understand the IR and see the actual shapes in the AST to which a lanague construct is mapped to it is of limited use if different parts of the tree need to be imploded and exploded to compare operations on programs with more than 500 lines of code. As an alternative visualization we use a PDF file, specifically, the PDF bookmark structure to represent the AST structure. For each node a bookmark and a single page showing all information that was generated at this node during an attribute evaluation.

Note that outputs can be generated at any point of a transformation. If a transformation is composed from several transformations, using higher-level abstractions to refine the transformations itself, an output after each step of a transformation can be generated and compared when debugging the code.

For both, PDF and DOT output, the output can be customized by the user by overriding a virtual function that is called during the generation at each node and allows to add user-defined information.

Finally, the generated source code itself can also be used for debugging because we preserve the original source structure beyond the C++ level. Preprocessor directives can be unparsed as well, allowing the user to see a source code that is not extended by hundreds of pages of declarations from `#include` directives. Although this detail has not got much attention by the community so far, it does help in the process of debugging.

Related Work

Within ROSE we use *Sage III*, which we have developed as a revision of the *Sage II* [4] AST restructuring tool. Work on *OpenC++* [5] has led to the development of a C++ tool which is also similar to Sage, but adds some additional capabilities for optimization. It neither addresses the sophisticated scale of abstractions that we target nor the transformations we target.

Related work on the optimization of libraries on *telescoping languages* [6] shares many of the same goals as our research work and we expect to work more closely with these researchers in the near future. Our approach so far is less ambitious than the *telescoping languages* research, but is in some aspects further along, though currently specific to high-level abstractions represented in C++. Further approaches are based on the definition of library-specific *annotation languages* to guide optimizing source code transformations [7] and on the specification of both high-level languages and corresponding sets of axioms defining code optimizations, see [8] for example.

Work at University of Tennessee has lead to the development of *Automatically Tuned Linear Algebra Software* (ATLAS) [9], within this approach numerous versions of a parameterized transformation are generated to define a search space and the performance of a given architecture is evaluated. The parameters associated with the best performing transformation are thus identified empirically. Preprocessors built using the ROSE framework could take significant advantage of this

```
int main ()
{
  int s=10;
  while (s>0) {
    s--;
  }
  return s;
}
```

Figure 4: A visualization of the AST. Such visualization can be generated for any subtree.

approach to identify the optimal parameters associated with transformations parameterized on architecture specific details (e.g. cache-size).

The example shown is similar to what can be done using expression templates techniques [10]. Expression templates work by using the template mechanism defined by C++ and requires no additional preprocessing step, but is exceedingly problematic. The general compile-time approach we present is superior, or at worst an alternative, because it provides for sophisticated program analysis which our simple example does not require, but more complex transformations (e.g. loop fusion) most certainly require. Such program analysis (e.g. dependence analysis) is not possible within the expression template technique because of the limitations inherent in template meta-programming. More information about expression templates and its advantages, disadvantages, and limitations can be found in [10–12].

Conclusions

We have presented a source-to-source architecture and its implementation for C++. The architecture allows to reinvoke the front-end to translate a source-fragment to an IR-fragment. This is possible because the source-fragmented is automatically extended by the Fragment Concatenator to form a complete program that can parsed by the front-end. The Fragment Extractor extracts an IR-fragment that corresponds to the source-fragment. Eventually this IR-fragment is inserted into the IR where specified. Invoking the back-end allows to generate a source-string for any IR-fragment. Both can be combined whereever the one or the other simplifies the definition of a transformation.

The attribute evaluation scheme in combination with reinvoking front-end and back-end allows to define a transformation as a side-effect free operation on the AST in C++. A transformation is the behaviour of an object and transformation hierarchies can be built by inheritance. Transformations can be applied in sequence and computed values for each AST node can be combined by using AST attributes which can be attached to any AST node.

Transformations are defined in C++, the same language in which the code is written which the user desires to optimize. The AST can be visualized as a tree in different formats like postscript, gif, or jpeg. And it can be viewed as PDF file, representing the tree structure in the PDF bookmark mechanism and all information about attributes and internal data (optional) for each node on a separate page.

The architecture allows users to define optimizations which the vendor compiler they are using does not perform for the abstractions they have built. Because such transformations are defined as source-to-source transformations they are best suited for high-level optimizations and the vendor compiler can still be used for low-level optimizations for specific architectures.

References

[1] Daniel Quinlan, Brian Miller, Bobby Philip, and Markus Schordan. Treating a user-defined parallel library as a domain-specific language. In *16th International Parallel and Distributed Processing Symposium (IPDPS, IPPS, SPDP)*, page 105. IEEE, April 2002.

[2] Daniel Quinlan, Markus Schordan, Brian Miller, and Markus Kowarschik. Parallel object-oriented framework optimization. *Concurrency: Practice and Experience*, to appear.

[3] Edison Design Group. http://www.edg.com.

[4] Francois Bodin, Peter Beckman, Dennis Gannon, Jacob Gotwals, Srinivas Narayana, Suresh Srinivas, and Beata Winnicka. Sage++: An object-oriented toolkit and class library for building fortran and C++ restructuring tools. In *Proceedings. OONSKI '94*, Oregon, 1994.

[5] Shigeru Chiba. Macro processing in object-oriented languages. In *TOOLS Pacific '98, Technology of Object-Oriented Languages and Systems*, 1998.

[6] B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, K. Kennedy, J. Mellor-Crummey, and L. Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. In *Journal of Parallel and Distributed Computing*, 2000.

[7] Samuel Z. Guyer and Calvin Liri. An annotation language for optimizing software libraries. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 39–52, Berkeley, CA, October 3–5 1999. USENIX Association.

[8] Vijay Menon and Keshav Pingali. High-level semantic optimization of numerical codes. In *Conference Proceedings of the 1999 International Conference on Supercomputing*, pages 434–443, Rhodes, Greece, June 20–25, 1999. ACM SIGARCH.

[9] R. Clint Whaley and Jack J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). In ACM, editor, *SC'98: High Performance Networking and Computing: Proceedings of the 1998 ACM/IEEE SC98 Conference: Orange County Convention Center, Orlando, Florida, USA, November 7–13, 1998*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1998. ACM Press and IEEE Computer Society Press. Best Paper Award for Systems.

[10] Todd L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science, Berlin, Heidelberg, New York, Tokyo, 1998. Springer-Verlag.

[11] F. Bassetti, K. Davis, and D. Quinlan. A comparison of performance-enhancing strategies for parallel numerical object-oriented frameworks. In *Proceedings of the first International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE) Conference*, December 1997.

[12] S. Karmesin, J. Crotinger, J. Cummings, and S. Haney. Array design and expression evaluation in POOMA II. *Lecture Notes in Computer Science*, 1505:231, 1998.